Team Rubik’s Noob

“No Rubik's cube should go unsolved”

Table of Contents

Introduction

Rubik’s Noob is a project where the application will solve any unsolved Rubik’s Cube pattern. This will help guide users on how to solve a Rubik’s Cube and better help visualize the cube. In addition, it will provide multiple solutions towards solving the Rubik’s cube in as efficient a manner as the user would like.

Group Members

Ryan Bell

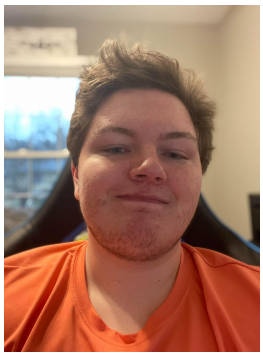
- **About Me:** Junior at the University of Missouri, majoring in CS with a minor in Math
- **Interests:** Sports, Video Games
- **Experience:** Unity, C#, C, HTML, JavaScript, Python
- **Role:** Unity Team

Tyler Wilkins

- **About Me:** Senior at Mizzou, majoring in CS with a minor in Math
- **Interests:** Video Games, Reading, Philosophy
- **Skills:** Java, C, C#, Unity, HTML, CSS, PHP, JavaScript
- **Role:** Algorithms Team, specializing in LBL and CFOP

Noah Backman

- **About Me:** Senior CS Major with IT Minor
- **Interests:** Enjoys a good cup of coffee, gaming, long walks on the beach
- **Skills**: C, Python, Java, Swift, and C#, and AI related algorithms
- **Role:** Algorithms Team, specializing in Thistlethwaite's Algorithm



Noah McDonell

- **About Me:** Senior at Mizzou, Dual majoring in Math/CS.
- **Interests:** Math, Gaming, Programming various things.
- **Skills:** Good with C, C++, C#, Java, Unity.
- **Role**: Algorithms Team, specializing in cube logic

Andrew Li

- **About Me:** Senior in CS
- **Interests:** Gaming, reading, finance, music
- **Skills:** Python, C++, Java
- **Role:** Algorithms, specializing in cube logic and overall algos

Jaylen Thomas

- **About Me:** Senior at Mizzou, Majoring in CS
- **Interests:** Gaming, Music, and Sports
- **Experience:** Python, Java, C, Mobile/Web Development Frameworks
- **Role:** Unity Team

Mission Statement

Solving a Rubik’s cube for the first time can be difficult. Despite this, a person over time can learn a single algorithm that will solve a Rubik’s cube comfortably. However, finding the most optimal algorithms based on the Rubrik’s cube’s patterns proves to be a challenge. That is where the project developed by the group Rubik’s Noob comes in. Our motto is that no Rubik's cube should go unsolved. We stand by this and we want people to be able to give their Rubik’s cube to our program and have it find the optimal way in solving it.

Mentor

The group will have Gary McKenzie as the mentor for this project as his expertise in the field of Artificial Intelligence and Algorithms would be beneficial to our development process.

Problem

Problem Definition

Many people struggle to figure out how to solve a Rubik’s Cube, which can be very frustrating, and the algorithms that are provided online can be challenging to follow.

Problem Solution

Rubik’s Noob will allow a user to input their unsolved cube, and the program will give them a step by step guide to solve it with a variety of different methods.

Benefits of Solution

Rubik’s Noob can be used by anyone who needs to unscramble a scrambled Rubik’s Cube for any reason, and could be used to teach solvers efficient methods to solving a cube.

Requirements

Hardware Requirements

- Users should have a scrambled Rubik's Cube on hand
- A computer running Windows or MacOS
- A computer capable of running the algorithms to solve a Rubik’s Cube (the requirements for Unity cover this as well)
- The project will be implemented in Unity, therefore hardware capable of supporting a Unity application will be required.

Minimum Unity Requirements [1]

Minimum Requirements	**Windows**	**MacOS**	**Linux**
OS Version	Windows 7 (SP1+) and Windows 10, 64-bit versions only	High Sierra 10.13+	Ubuntu 16.04, Ubuntu 18.04, and CentOS 7

CPU	X64 architecture with SSE2 instruction set support	X64 architecture with SSE2 instruction set support	X64 architecture with SSE2 instruction set support
Graphics API	DX10, DX11, and DX12-capable GPUs	Metal-capable Intel and AMD GPUs	OpenGL 3.2+ or Vulkan-capable, Nvidia and AMD GPUs.
Additional Requirements	Hardware vendor officially supported drivers	Apple officially supported drivers	Gnome desktop environment running on top of X11 windowing system, Nvidia official proprietary graphics driver, or AMD Mesa graphics driver. Other configuration and user environment as provided by default with the supported distribution

User/Non-User or Functional/Non-Functional Requirements

- Users should be able to easily and intuitively input the colors from their scrambled Rubik's Cube into the program.
- Users should be presented with a clear visualization of the Rubik’s cube
- Users should intuitively be able to tell how to use the application and how to understand given solutions.
- Users should be able to choose from multiple approaches to how their cube should be solved
- Users should be informed as to the differences involved in each solving algorithm

- Users should be able to obtain solutions to their Rubik’s Cube in a short amount of time.
- Users should be able to step-by-step through the generated solutions virtually
- Users should be able to understand the moves being displayed and how to follow along
- Users should be able to step backwards through the solution in case a step is misunderstood
- Users should also be able to play a complete solution animation rather than going step by step

Project Research

Project Research Introduction

The two main topics to discuss and ponder are the algorithms and the software for this project. These are broken down into six research topics: best languages to use for solving the cube, best software to use for visualizing the cube, representing the cube in software, solving methods for the cube, algorithmic approach towards solving the cube, and brute force algorithms for solving the cube.

Best Languages to Use for Rubik's Cube Solving

There are an abundant amount of coding languages available, especially for a Rubik's Cube solving project. However, a single programming language must be chosen for the algorithmic backend. The three languages being considered are C++, Java, and Python. There are three factors to consider in order to choose a primary coding language: libraries and resources, computing time, and compatibility. With these three factors in mind, Python is the language of choice for this project on the algorithmic side.

It is essential to not reinvent the wheel when it comes to programming an application. Utilizing available resources and libraries is an important part of software development. With a quick google search, there are more Python resources available for a Rubik's Cube than Java and C++. Here are some Python libraries available: pytwisty, rubik-solver, pglass.These libraries can provide guidance as to how to represent our

cube in a backend level. Furthermore, the mechanics of moving the cube can already be solved, allowing us to focus solely on the algorithmic side of solving a cube.

C++ is very fast and utilizes less memory than Java and Python. In a study regarding speed and performance with languages parsing bioinformatics data in a Windows and Linux OS, C++ is the fastest while Python is the slowest and memory heavy [2]. However, given the cube’s information, cube solving performance on the coding and host OS level is negligible due to the execution environments matching. For instance, if the user runs the application, the program’s algorithms will only run in one primary language, thus comparing times of solving the Rubik’s cube rather than the time of execution since it will be similar among all tests. To mitigate potential outliers, there can also be many Rubik’s cubes being solved at once with a specific algorithm which will allow for a more average accurate time. Although the language’s performance and memory usage should be considered, it is not the decisive factor in choosing a language.

Some considerations as to how to visualize the cube have been vocalized. Two visualization methods to consider are through the terminal or through Unity. C++, Java, and Python can display their output and cube visualizations through the terminal feasibly. However, compatibility with Unity is more difficult. C++ is not the primary language for Unity but its similar counterpart C# is available. Python is also not a primary language in Unity but there is support for its compatibility [3]. Since this is heavy

on the algorithmic side of programming, Tensorflow is something to consider. In that regard, Python is very much compatible [4].

C++	**Java**	**Python**
Fastest, low memory	Faster, medium memory	Fast, heavy memory
Unity	No Unity	Unity
Tensorflow	No Tensorflow	Tensorflow
Few resources	Few resources	Abundant resources
Good visualizations	Better visualization	Best visualizations

With all of these considerations in mind, the primary language of choice for the backend development of this Rubik's cube solving application will be Python. Despite its slow execution time and heavy memory usage, Python's amount of resources and compatibility with possible other software are the most important considerations.

Best Software to Use

When discussing 3D rendering, a few names quickly come to mind: Unity, Blender, 3DS, etc. However, there are plenty of other alternatives out there that can provide a similar experience and better suit the needs of this project. For the scope of this project, we mainly want to look into 3 things when determining what rendering software works best for us; that is cost, hardware overhead, and ease of use. Of course there are other determining factors that will be accounted for, but these are the frontrunners for our purposes.

Unity

Unity is a well-known and widely-used 3D rendering software that's main goal is to provide a platform for creating Real-Time 3D Content [5]. Although their focus was game development for many years, they have branched off into multiple other industries such as automotive manufacturing, film, and architecture. What are the pros and cons of using Unity for our purposes?

Cost

Unity offers 4 plans for potential users; these are Personal, Plus, Pro and Enterprise. Since Personal is the only tier that works within our budget, that is the only plan that will be touched on. Personal offers the bare minimum with access to Unity’s development platform, one Integration with collaboration tools, 3 seats with Unity Teams, and Unity Ads to monetize. This is the free alternative. Unity also offers a Student Plan, available to students enrolled in accredited institutions. This will allow you to hold 5 seats in Unity Teams Advanced, gain additional cloud storage, and “use the same tools and workflow that professionals use” according to Unity writer Nichole Yong. This will probably do for the needs of our project.

Hardware Overhead

Unity 2020 requires that you have at least Windows 7 for Windows OS and High Sierra 10.13+ for macOS [5]. CPU-wise you will need X64 architecture with SSE2 instruction

set support for both Windows and Mac. As far as Graphics APIs you will need anything DX10+ capable GPUs for Windows users, and Metal-capable Intel and AMD GPUs for Mac users. There are generally no additional requirements for both OSs. It is clear that these are the baseline requirements, but there is no guarantee that you can run just anything with these baselines. Unity General Support mentions that while 4GB of RAM can run Unity you will be limited to 2D or low poly 3D rendering. Our project requires 3D rendering, and while it might not be the most intensive of renderings, we still want to be able to work on the project as efficiently as possible. This can’t be done if it takes hours for something that is described as a low poly rendering to complete. In the case of Unity, our baseline is more along the lines of 8GB of RAM, preferably 16GB if possible.

Ease of Use

There are multiple ways to approach the problem of representing the Rubik's Cube in Unity, each with its own pros and cons. There are plug-ins for Unity such as Meshroom that use photogrammetry to create 3d models from objects you’ve scanned [6]. This would take a shorter time than creating a model by hand, but there has to be a way to dissect this model and make it solvable. In the article it also states that the scan requires a fairly powerful CPU to churn through the work in a reasonable amount of time. This won’t be a problem if using a powerful machine, but we will have to gain access to said machine. Otherwise, similar to Jon Worthington’s implementation, we can have 27 separate cube models all to represent the full cube and make it easier to work with. With that being said, Unity provides a solution for multiple options, it will all

come down to what path we choose to take. As far as Programming languages goes, Unity natively supports C# and a few other .NET languages.

Blender

Blender is the free and open source 3D creation suite that supports the entirety of the 3D pipeline - modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation [7]. It allows for Python scripting to customize the application and is well suited for individuals and small studios.

Cost

Like stated above, Blender is free to use. Some of its features include rendering, modeling, simulation, scripting, and more. However, if you would like to use the Blender Cloud, a source for assets and training, it’ll cost 10$ per month. This shouldn't be a necessity, as most of the assets are readily available for free.

Hardware Overhead

Blender has a much higher hardware overhead to use. The minimum requirements include having at least Windows 8 or Mac 10.13, a 64-bit quad core CPU with SSE2 support, 8GB of RAM, a full HD display, and a Graphics card with 2 GB RAM. The recommended requirements are a 64-bit 8 core CPU, 32 GB of RAM, a 2560 x 1140 display, and a Graphics card with 8 GB of RAM. The average machine will not have the

recommended specs to use blender, so we may have to find a machine capable of meeting these standards.

Ease of Use

Similar to in Unity, Blender can represent block components of the Rubik's Cube using Child Of Constraint and rotation bones. This is of course the straightforward path, with other options such as drawing the Cube using OpenGL command in Python or using something from the GUI. Also, Blender uses Python as its primary programming language, something that most of our group members have mentioned they are comfortable with.

In conclusion, we’ve come to the decision to use Unity. Unity will allow for less trivial animation because at its core, it is a game engine. We will also be able to use Python Scripting to access in-engine objects instead of relying solely on C#.

Representing Rubik’s Cubes in Software

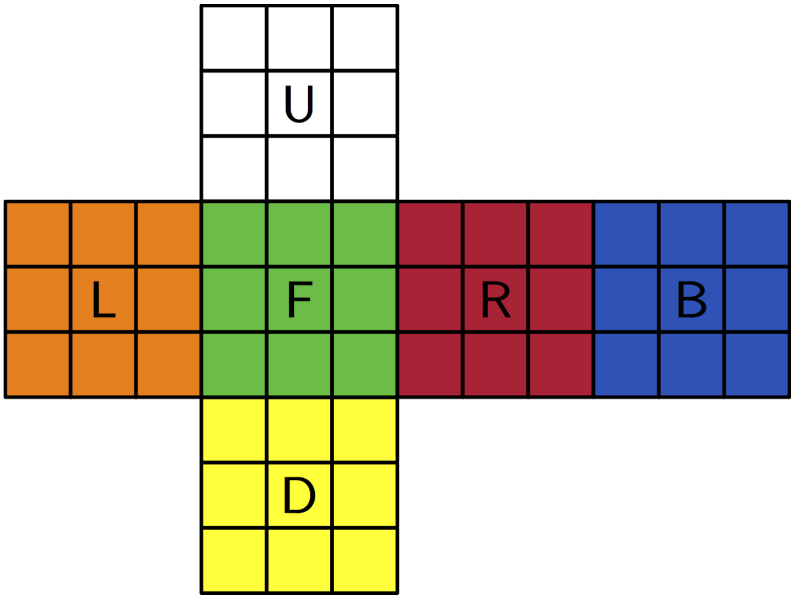
In order to simulate a Rubik's Cube on a computer, we need a digital representation of the cube that can be manipulated with functions. In theory, representing a Rubik's Cube in code should be simple. It is a cube with 6 faces, each face containing 9 colored squares. We can represent the cube simply as a 2-dimensional array.

Cube[x][y] = Color of the yth cell on the xth face.

We can order the cells in each face as follows

0	1	2
3	4	5
6	7	8

This is not enough to accurately represent our cube. Because a Rubik's Cube face can be oriented in any direction, how do we know which way a face is rotated? We can fix this problem by modeling the cube as a flat projection



And because the positions of the center colors do not change, we can assign the faces as follows:

- Front: Green
- Up: White
- Left: Orange
- Down: Yellow
- Right: Red
- Back: Blue

More details and a complete solution to the problem is located in the Project Design Specifications section of this documentation.

Introduction to Rubik's Cube Solving Methods

This project will include a feature that will solve a scrambled Rubik's Cube using well-established methods often used in the real world. There are several of these methods that exist, each with their own complexities, advantages, and disadvantages. Below is a summary and comparison of some common methods that were considered before deciding on which method to use [8].

Layer By Layer Method

The Layer-by-Layer Method, also known as the beginner method or as LBL, is often the first method someone new to Rubik's Cubes will learn. As the name implies, this method works by solving the cube one layer at a time. LBL is made up of about five steps and requires knowledge of only a few basic sets of moves. However, this method is far from efficient as it takes approximately 130 moves on average to solve a cube this way [9].

CFOP Method

The CFOP Method, also known as the Friedrich Method, is a more advanced form of the LBL method. Where LBL solves the cube one layer at a time, CFOP solves two of the cube's layers in a single step, leaving the rest of the method to solve the last layer. In the acronym CFOP, each letter stands for one of the method's four steps. These are Cross (C), First Two Layers (F), Orientation of the Last Layer (O), and Permutation of the Last Layer (P). CFOP is a widely popular algorithm in speed cubing for its efficiency and its being easy to learn. CFOP has statistically demonstrated itself to be the fastest

method, it is the most researched, and does not require the solver to have an expert knowledge of how a Rubik's Cube works. While this method has an average solve of approximately 60 moves, it requires the memorization of 78 sets of moves [10]. So, while this method is efficient in terms of moves, it is balanced by the number of algorithms required.

Roux Method

The Roux method approaches solving a Rubik's Cube in a unique way. Rather than thinking of the cube as a series of layers to be solved, the Roux method uses a method called blockbuilding to solve the corners of the Rubik's Cube first. This allows for free use of the vertical middle slice of the cube without having to worry about scrambling already solved portions of the cube. This method was created in an attempt to minimize moves and provide an efficient way to solve a Rubik's Cube with only one hand. This method solves a Rubik's Cube with 4 steps, an average of about 50 moves, and requires the memorization of 42 move sets [11]. While this method is efficient, it is considered difficult as it is not very intuitive and efficiently rotating the middle slice of a cube can be difficult. Given its difficulty and the fact that its moves are far from intuitive, this method is usually only utilized by those with experience with other methods already.

The ZZ Method

The ZZ Method is the most modern method considered here, having been created in 2006. ZZ stands for the initials of the methods creator, Zbigniew Zborowski. The ZZ method consists of three steps and uses a combination of block building and LBL

methodology. ZZ was created in an effort to create a method which allowed people to solve the cube using easier movements, allowing for a faster solve time. This method has several variations, each of which includes a slightly different take on the three basic steps and includes different sets of moves. The ZZ method can solve a Rubik's Cube with an average move count of about 45 moves. However, depending on the variation of ZZ utilized, this method could include the use of up to 514 different move sets [12]. Similar to the Roux method, this is a more advanced method and is more difficult to grasp than other methods.

Method Comparison

	Step Count	Move set Count	Average Move count	Overall Difficulty
LBL	5	6+	130	Very Easy
CFOP	4	78	60	Easy
Roux	4	42	50	Very Difficult
ZZ	3	514	45	Difficult

Conclusion on Rubik's Cube Solving Methods

Given the above summaries and comparisons of common methods to solve a Rubik's Cube, it is important to list the criteria by which a decision will be made. First the algorithm must be made up of a reasonable number of steps, a given solution must be made up of a small enough list of moves that a user can easily follow, the move set count must be reasonable for programming purposes, and the difficulty must be reasonable such that a user wanting to learn how to solve a Rubik's Cube could understand the moves being made. Given these criteria, LBL and ZZ will be eliminated first as LBL has an unreasonable average move count and ZZ has an unreasonable move set count. This leaves the CFOP and Roux methods. Both of these methods have the same step count and similar average move counts, although Roux is slightly more efficient. Additionally, Roux has the advantage of necessitating the knowledge of a fewer number of move sets, at only 42 versus CFOP's 78. However, given the purpose of this feature is to help users understand and learn how to solve a Rubik's Cube, CFOP's popularity and more intuitive and easier to learn nature makes it the most user-friendly choice. For this reason, CFOP is the method which will be used for this feature to demonstrate a common and efficient method of solving a Rubik's Cube.

Decision Tree Based Approaches (not usable by humans)

While the other two algorithms being investigated will be relatively easy for a human to reproduce, it's also important to investigate an algorithm that will deliver a near optimal solution for solving a Rubik's Cube – even if the method is too complex for a human to replicate. In order to do this, an algorithm with a decision tree needs to be used. A

decision tree consists of nodes that describe a state of the cube. Branches from the node lead to nodes/states that can be reached by making a move with the cube.

An important aspect of the algorithm is how it traverses the tree. The two most fundamental ways of traversing a tree are Breadth First Search and Depth First Search. Without going too in depth (haha get it), Breadth First Search prioritizes exploring each possible move from each node explored whereas Depth First Search will prioritize making as many moves down the tree as possible before backtracking and trying others. There are variations on these, however.

Importantly, to evaluate the viability of a move to a new state, the algorithm must be able to see how far the cube would be from being solved in that state. This is where heuristics come into play. A heuristic estimates how far the cube is for its completed state. However, in order for the heuristic to be admissible, it needs to not overestimate the number of moves necessary to reach the completed state.

Korf's Algorithm

One possible algorithm that uses a combination of heuristic search and a variation of Depth First Search is Korf's algorithm. Korf's algorithm was developed by Richard Korf in 1997 and is designed to find an optimal solution for a randomized Rubik's Cube (the maximum number of moves to a solution using this algorithm is 20). This algorithm implements a combination of IDDFS (Iterative Deepening Depth First Search) and A* search called IDA* (Iterative Deepening A*) [13]. Because of the difficulty of calculating a heuristic for every state of the cube, a database of heuristics is implemented to make

it easier to calculate the viability of a state. However, due to the exhaustive nature of finding the most optimal solution for a given cube, the algorithm takes a LONG time to find a solution and wouldn't be feasible for a program that teaches someone how to solve a cube. [14]

Thistlethwaite's Algorithm

So while the OPTIMAL solution may be out of the question, we can still find a great solution in far less time by simply lowering our standards. This is where Thistlethwaite's algorithm comes into play. Like Korf's algorithm, Thistlthwaite's uses IDFS search but reduces the branching factor by dividing the path to the solution into groups. What these groups do is limit the amount of moves that can be taken until a new group is reached, which heavily reduces the branching factor. Now in doing this, the maximum number of moves to reach a solution gets closer to 50, but the algorithm in turn is much faster. [15] This is a much more feasible algorithm to use for our program.

Brute Force Rubik's Cube Solution

There needs to be a method which can be used as a baseline to compare other algorithms against. At first a "brute force" method seemed like a good baseline algorithm. Since a Rubik's Cube has a finite number of possible combinations, there is a "brute force" method which could solve a Rubik's Cube from any given starting position. In theory, there is a sequence of moves that would be able to cycle through all of the possible combinations of a Rubik's Cube, so the "Brute force" method would be able to reach a completed cube from any given position. However, there are over 43 quintillion

(43,252,003,274,489,856,000) possible combinations in a 3x3 Rubik's Cube [16]. Thus, this sequence of moves, referred to as "The Devil's Algorithm" would be extremely long and has yet to be found for a 3x3 Rubik's Cube.

While there has not been a true "Brute Force" algorithm for a 3x3 Rubik's Cube, there has been some success in finding one for a 2x2 Rubik's Cube. In 2011, Cuber Bruce released a theory for the first "Devil's Algorithm" on a 2x2 Rubik's Cube. This algorithm used a Hamiltonian Circuit, a circuit that visits every combination and ends in the starting combination, to cycle through each possible combination of a 2x2 cube [17]. This algorithm had 800 thousand moves which could be repeated five times and returned to the starting position in the end [17]. So, even when using only a 2x2 Rubik's Cube, a "Brute Force" method of solving is lengthy and impractical.

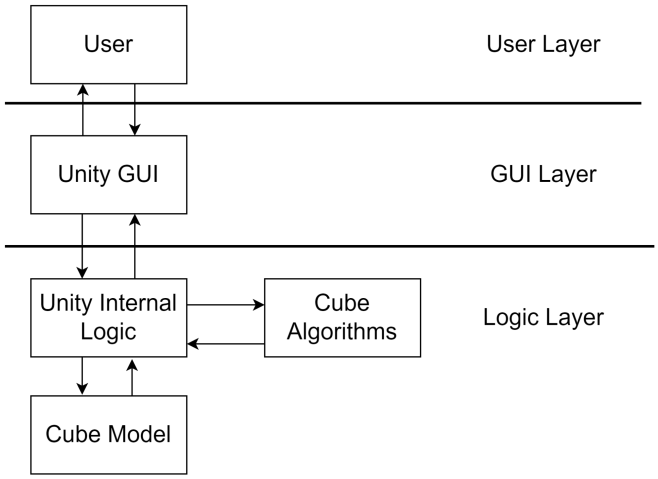
Since a true "Brute Force" method is out of the question, there are many algorithms that are very simple that we could use instead as a baseline. The Layer by Layer method is quite simple and will work with any given Rubik's Cube. This method is not very efficient, and it would be a good baseline to compare different algorithms against instead of the true "Brute Force" method [18].

Project Design Specifications

The following portion of the document will describe how this project will be implemented. The design has been broken up into individual sections, with each section describing the process to implement that portion of the project.

System Architecture Diagram

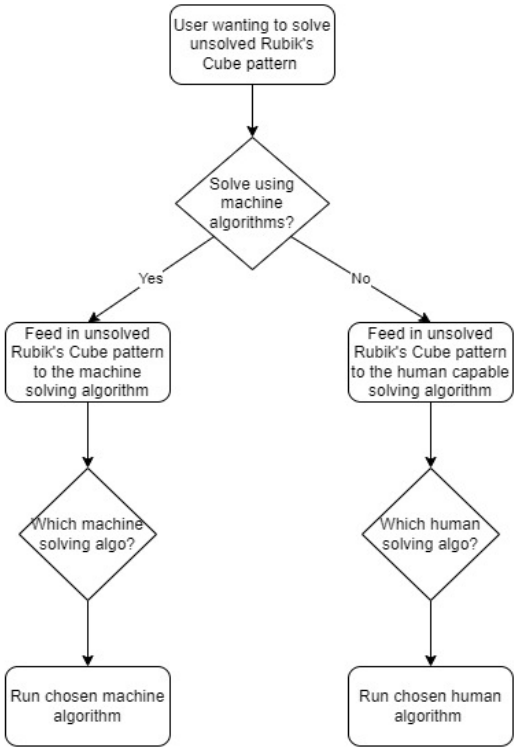
This system architecture diagram outlines the architecture of our program. The user will only interact with the Unity GUI. The GUI will interact with Unity's internal logic and C# scripts, which in turn will access and modify the internal cube model, and run cube solving algorithms to display the results to the user.



Decisional Flow of the Application System

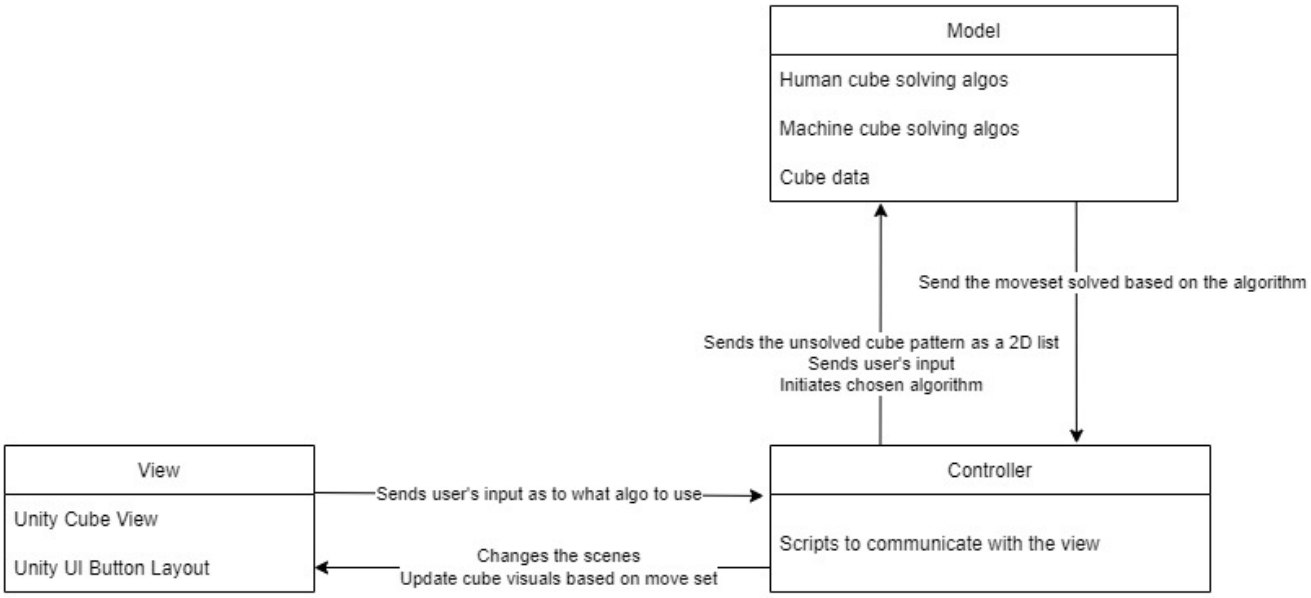
For this application, there will be no database since a majority of this project utilizes algorithms. The user will first send the Rubik's cube file to start the application. The user

will be able to choose what algorithm to use to solve the cube. After solving, the user can then reset and follow the same process.



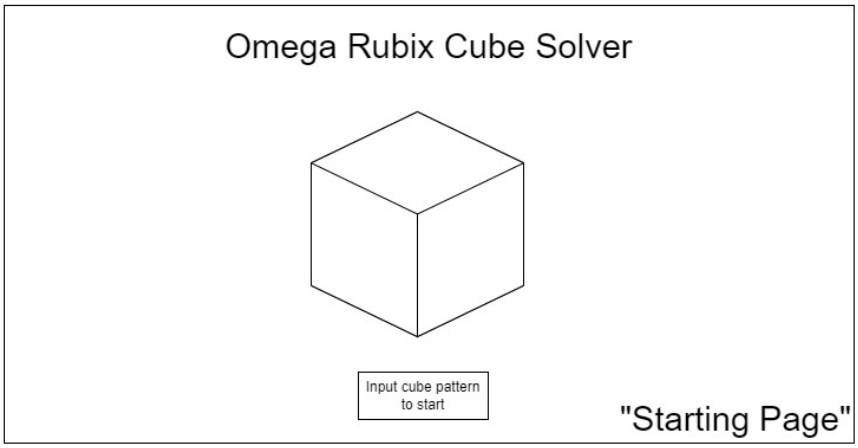
Framework Graph (MVC)

This application uses the Model-View-Controller architecture since this is mostly a client facing application. The user will make changes on the View which will then communicate with the Controller to send the input given to the Model. The Model will generate the moveset based on the selected algorithm and send that back to the Controller. The script in the Controller will be able to read the moveset and communicate with the View on how to update the scene.

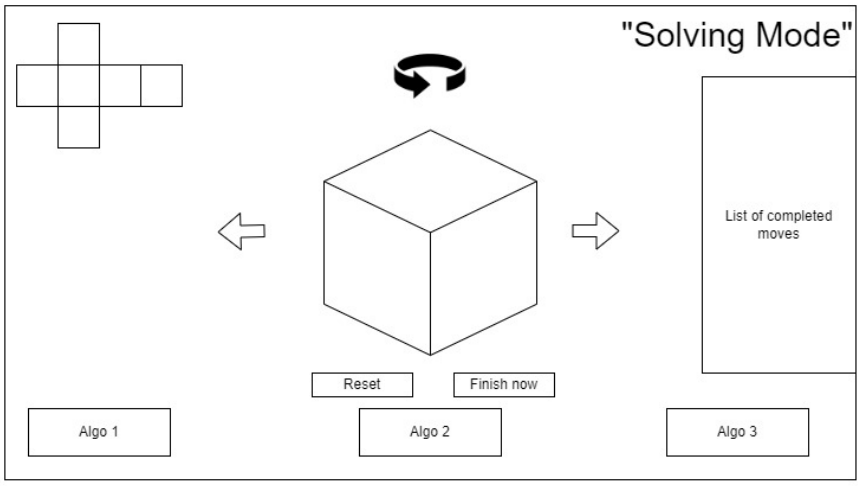

Model
Human cube solving algos
Machine cube solving algos
Cube data
Send the moveset solved based on the algorithm
Sends the unsolved cube pattern as a 2D list
Sends user's input
Initiates chosen algorithm
View
Unity Cube View
Unity UI Button Layout
Sends user's input as to what algo to use
Changes the scenes
Update cube visuals based on move set
Controller
Scripts to communicate with the view

Graphic User Interface

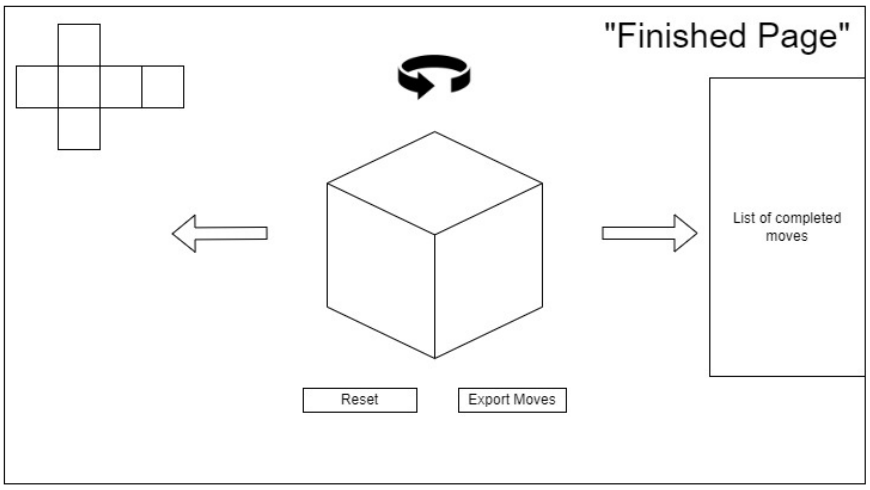
Here is our Graphic User Interfaces that we have created. We have the Starting Page which is where the user will input the Rubik's Cube as a .txt file.The program will parse this information to set up the initial cube state. The user will only have to input the .txt file as the Rubik’s Cube pattern. Filling in this file will be discussed in a later section (Inputting Cube Pattern).



We will have a Solving Mode which is where the user chooses which algorithm to use to solve the given Rubik's Cube. The user can go step by step through each of the moves, go back to the move before, or choose to finish the algorithm quickly.

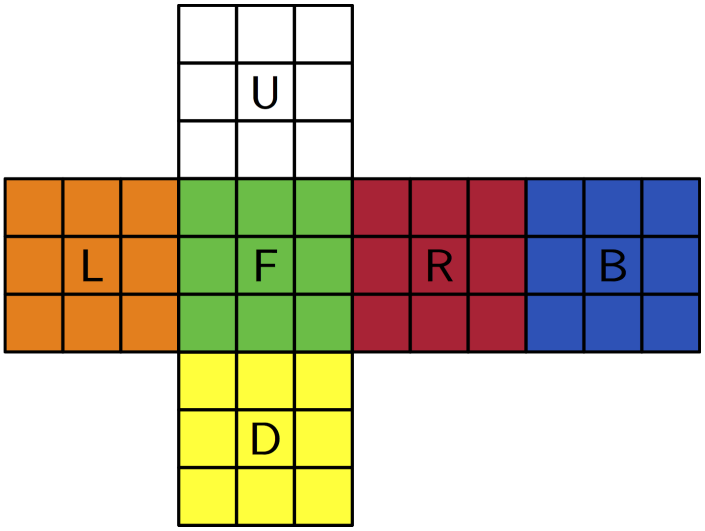


Lastly, we will have a Finished Page that will show the completed Rubik's Cube. This GUI will display the list of moves to solve the Rubik's Cube with the selected algorithm. There is also an option to export the moves into a file, or you can reset to solve the cube with a different algorithm.



Inputting the Cube Pattern

The user will be prompted in the starting page to input the Cube's patterns. This will bring up a prompt dialog where the user can input the colors on the cube's face from left to right, top to bottom. Each line will represent a face of the cube. The order will proceed as follows: upper, right, down, left, front, back. For instance, the image below is a solved cube.



The user will input this cube pattern as the following:

WWWWWWWWW

RRRRRRRRR

YYYYYYYYY

OOOOOOOOO

GGGGGGGGG

BBBBBBBBB

In order for the cube to be completely solved, this is the format the application must follow. Otherwise, it is not solved correctly. There will be checks to ensure that there is no invalid color or an invalid completed face since the middle color cannot be changed. Furthermore, there will be a disclaimer to the user for what order the colors must be inputted as.

Representing Cube in Code, Pseudocode Functions for Cube Manipulation

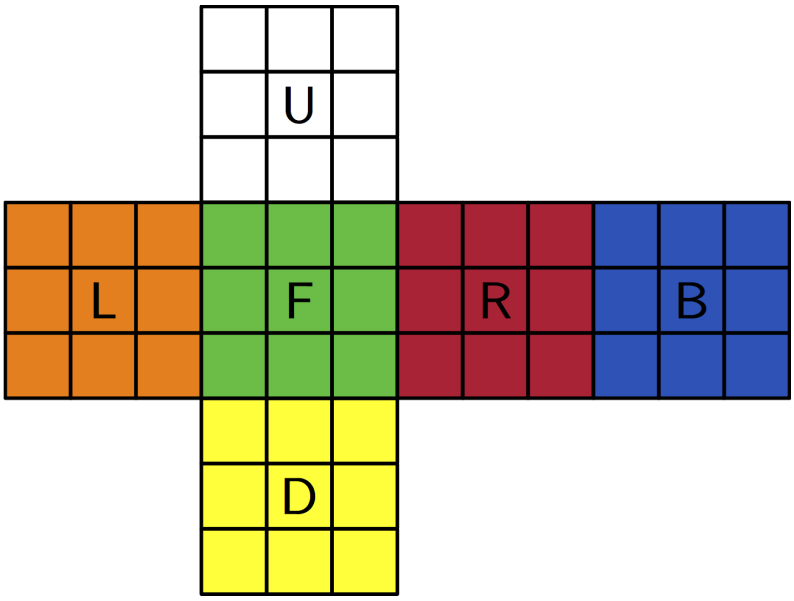
In order to simulate a Rubik's Cube on a computer, we need a digital representation of the cube that can be manipulated with functions. In theory, representing a Rubik's Cube in code should be simple. It is a cube with 6 faces, each face containing 9 colored squares. We can represent the cube simply as a 2-dimensional array.

Cube[x][y] = Color of the yth cell on the xth face.

We can order the cells in each face as follows

0	1	2
3	4	5
6	7	8

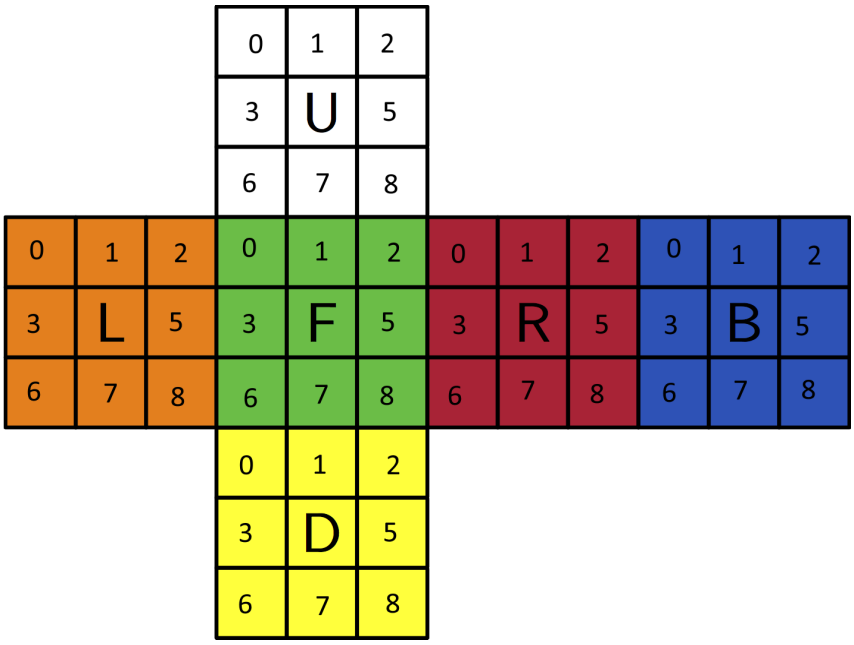
This is not enough to accurately represent our cube. Because a Rubik's Cube face can be oriented in any direction, how do we know which way a face is rotated? We can fix this problem by modeling the cube as a flat projection



And because the positions of the center colors do not change, we can assign the faces as follows:

- Front: Green
- Up: White
- Left: Orange
- Down: Yellow
- Right: Red
- Back: Blue

We can order the faces in the array in this order. And for each face on the flattened projection, we can assign cell numbers from top-down left to right.



So with this model, we have an accurate digital representation of a Rubik's Cube. If we want to perform operations on the cube, such as rotations.

For example, if we wanted to rotate the front face right, we would reassign the cells as follows

Cells [0,3,6] of R would get reassigned to cells [6,7,8] of U

Cells [0,1,2] of D would get reassigned to cells [0,3,6] of R

Cells [2,5,8] of L would get reassigned to cells [0,1,2] of D

Cells [6,7,8] of U would get reassigned to cells [2,5,8] of L

And for the front face, we would reassign the cells as follows

0->2, 1->5, 2->8, 3->1, 4->4, 5->7, 6->0, 7->3, 8->6

If we wanted to rotate left, we would simply reverse these operations, and if we wanted to rotate any other face, we would just recompute these transitions. This model gives a complete representation of a Rubik's Cube, and allows us to perform rotation operations on it.

We can write pseudocode to implement these functions. We must first create two enumerations to encode each face of the cube, and the color on each face.

```
enum faces{
      FRONT = 0,
      UP,
      LEFT,
      DOWN,
      RIGHT,
      BACK
}

enum colors{
      GREEN = 0,
      WHITE,
      ORANGE,
      YELLOW,
      RED,
      BLUE
}
```

Now, we can write the functions. This first function will initialize a cube with default values.

```
function initializeCube(cubeArray):
      for face = 0 to 5:
            for cell = 0 to 8:
                  cubeArray[face][cell] = colors[face]
```

The next function takes in a cube array as defined above, and performs a left rotation of the front face of the cube by reassigning groups of colors.

```
function rotateFrontLeft(cubeArray):
      upCells <- cubeArray[UP][6,7,8]
      rightCells <- cubeArray[RIGHT][0,3,6]
      downCells <- cubeArray[DOWN][0,1,2]
      leftCells <- cubeArray[LEFT][2,5,8]
      cubeArray[RIGHT][0,3,6] <- upCells
      cubeArray[DOWN][0,1,2] <- rightCells
      cubeArray[LEFT][2,5,8] <- downCells
      cubeArray[UP][6,7,8] <- leftCells
      frontCells <- cubeArray[FRONT]
      cubeArray[FRONT][2] <- frontCells[0]
      cubeArray[FRONT][5] <- frontCells[1]
      cubeArray[FRONT][8] <- frontCells[2]
      cubeArray[FRONT][1] <- frontCells[3]
      cubeArray[FRONT][7] <- frontCells[5]
      cubeArray[FRONT][0] <- frontCells[6]
      cubeArray[FRONT][3] <- frontCells[7]
      cubeArray[FRONT][6] <- frontCells[8]
```

For all of the other rotation operations, the function will be mostly the same except for swapped faces and cell numbers.

How the Layer-By-Layer Method will be Implemented

The Layer-By-Layer (or LBL) method was chosen as the method which will be used as the basic Rubik's Cube method to be implemented, as explained in the research portion of this document. LBL is a method that consists of six steps, forming a white cross, solving the white corners, solving the second layer, forming a yellow cross, and solving the yellow corners. Each of these steps consists of sets of moves called algorithms

which are used to get the cube ready for the next step. Each of these move algorithms use an "if-then" approach such as, "if pattern a exists on the cube, perform algorithm x".

LBL will be implemented by dividing each step of the method into its own function. Each of these functions will take the array representing the Rubik's Cube and a moveList array of characters recording completed moves, as input. The moveList will be unique for each scrambled cube, and will be dynamically defined so as to allow for how many moves are necessary. The functions will then check the current state of the cube, identify which set of moves, or algorithm, is appropriate to perform next, perform the moves called for by that algorithm, and record the moves as characters in the passed character array. Once the cube is in a state that is ready for the next step of the method, the function will then return the cubeArray and moveList arrays. This will continue until all steps are completed and the cube has been solved.

Below is pseudocode for the main LBL solving function. Each function called in this function represents a step of the LBL method.

```
function solveLBL(cubeArray){
      INPUT: Array representing the Rubik's cube
      moveList = new character array
      whiteCross(cubeArray, moveList)
      whiteCorners(cubeArray, moveList)
      secondLayer(cubeArray, moveList)
      yellowCross(cubeArray, moveList)
      yellowEdges(cubeArray, moveList)
```

```
      yellowCorners(cubeArray, moveList)
      RETURN: cubeArray, moveList }
```

Below is a generic pseudocode, not specific to any one step of the LBL method, for how each of the step functions will operate.

```
function solutionStep(cubeArray, moveList){
      INPUT: Array representing the Rubik's cube, Array for moves
      WHILE cubeArray does not match the desired pattern
            IF cubeArray matches pattern for algorithm 1
                  algorithm1(cubeArray, moveList)
            ELSEIF cubeArray matches pattern for algorithm 2
                  algorithm2(cubeArray, moveList)
            ELSEIF cubeArray matches pattern for algorithm 3
            …
            ENDIF
      ENDWHILE
      RETURN: cubeArray, moveList
}
```

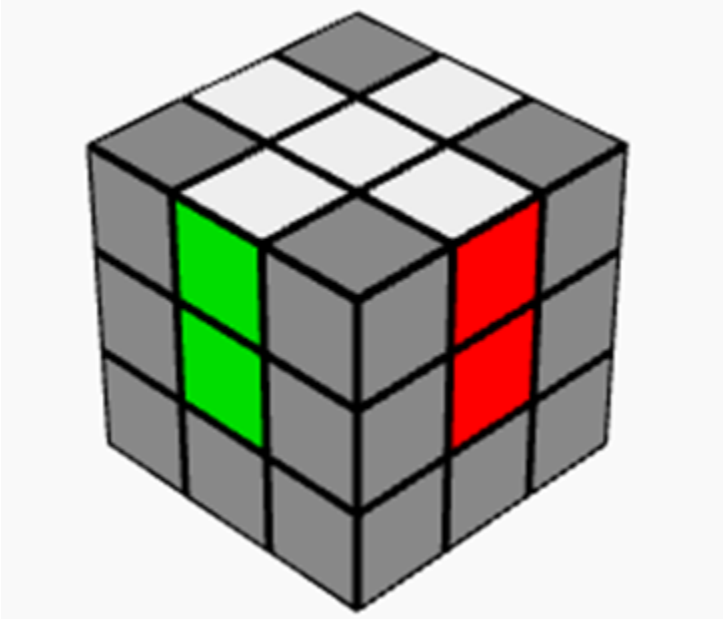
Similar to each of the steps in the LBL method, each of the move algorithms will be implemented as a function. These move functions will take the cubArray and moveList, move the cubeArray according to the algorithm, update the moveList, and return both the updated cubeArray and moveList. Below is generic pseudocode for these move functions.

```
function moveAlgorithm(cubeArray, moveList){
```

```
            INPUT:  Array representing the Rubik's cube, Array for moves
            firstRotation(cubeArray)
            update(moveList)
            secondRotation(cubeArray)
            update(moveList)
            thirdRotation(cubeArray)
            update(moveList)
            …
            RETURN: cubeArray, moveList
    }
```

Here is a short example of this logic working. In the case that the SolveLBL() function is passed a scrambled cube, the first thing the function will do is create a moveList character array. The cubeArray and moveList will then be passed to the first step function, whiteCross(). whiteCross() will then begin the process of identifying which algorithm needs to be performed based on the patterns present in the cube. For this example, assume the cube is in a state where the white cross will be formed after a single edge piece is flipped such as shown below[18]:

whiteCross() will then call the appropriate algorithm to complete the white cross based on this pattern. In this case, that algorithm is F, U’, R, U (where an apostrophe indicates counterclockwise rotation rather than clockwise). The move algorithm will append these moves to the moveList, then return both the cubeArray and the moveList. The whiteCross() function will then recognize that the goal pattern has been reached as shown below[18]:



whiteCross() will then return the cube and moveList to the solveLBL() function, which will continue moving through the steps until the cube is solved[18].

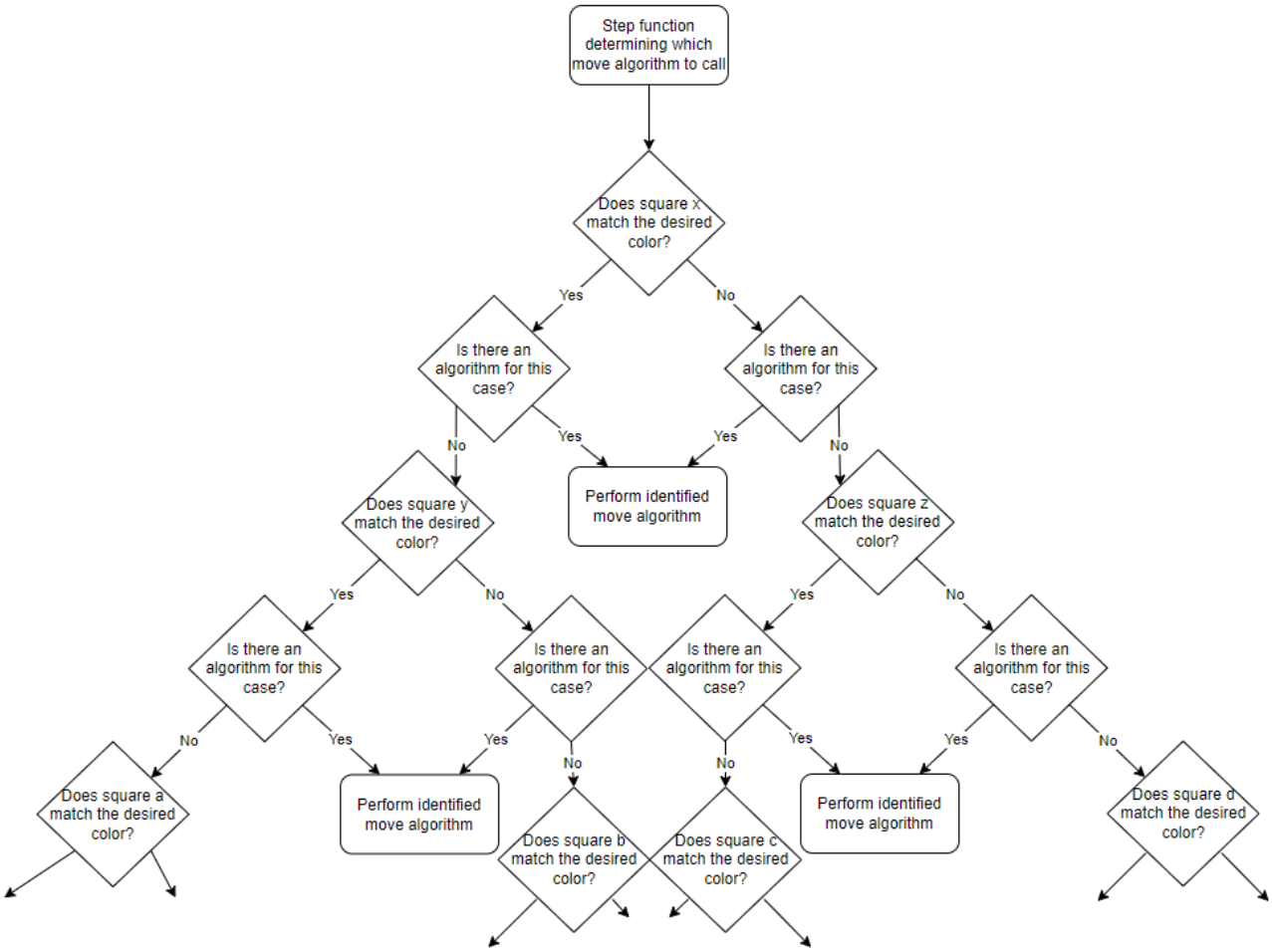
How CFOP will be implemented

The CFOP method of solving is logically very similar to the LBL method. Because of this, the implementation of CFOP will be very similar to that of LBL. However, rather than having six steps, CFOP has four: forming a white cross, solving the first two layers, orientating the last layer, and permutating the last layer. Similar to LBL, these four steps will be divided into individual functions as shown below:

```
function solveCFOP(cubeArray){
      INPUT: Array representing the Rubik's cube
      moveList = new character array
      whiteCross(cubeArray, moveList)
      firstTwoLayers(cubeArray, moveList)
      orientLastLayer(cubeArray, moveList)
      permutateLastLayer(cubeArray, moveList)
      RETURN: cubeArray, moveList
}
```

The primary difference in implementation of CFOP will be the number and complexity of the move algorithms. As explained in the research, CFOP achieves its efficiency by being composed of many more move algorithms than LBL. To account for this, the CFOP step functions will use decision trees to optimize the process of identifying which algorithm should be performed. These decision trees will check a relevant square of the cube, check if that square's color completes a pattern with an algorithm, then perform that algorithm if it exists or continue if not. The trees will continue by only considering

patterns which allow for the colors of the squares that have already been considered. Below is a flowchart demonstrating this process:



Apart from this change in how the step functions will identify which move algorithm to perform, the step functions and move algorithm functions will function in the same way as described for LBL. Because of this, and because both LBL and CFOP have the same first step, the example provided for LBL applies to this CFOP implementation as well.

Implementation of AI-Based Algorithm (Thistlethwaite)

Thistlethwaite's algorithm involves a breadth-first search of a decision tree of cube states. The algorithm is divided into phases, and each phase has an increasingly limited set of moves that can be made on each part of the cube. These moves are contained in applicableMoves, and are constant for every run of the algorithm.

Pseudocode:

```
INPUT: Starting cube state as currentState
cubeSequence = [currentState]
for phase in range(4):
    currentID, goalID = currentState.id(phase), goalState.id(phase)
    states = [currentState]
    stateIDs = set([currentID])
    if currentID != goalID:
        phaseComplete = false
        while(!(phaseComplete)):
            nextStates = []
            for workingState in states:
                cubeSequence.append(workingState)
                for move in applicableMoves[phase]:
                    nextState = workingState.applyMove(move)
                    nextID = nextState.id(phase)
                    if nextID == goalID:
                        phaseComplete = true
```

```
                        currentState = nextState
                        break
                    if nextID not in stateIDs:
                        stateIDs.add(nextID)
                        nextStates.append(nextState)
                if phaseComplete:
                    break
            states = nextStates
RETURN: Completed cube state and cubeSequence
```

Development Strategy

Since this project will be over the course of a semester, the development framework will follow an Agile Scrum framework. This framework was chosen because it entails that the workflow will follow in a sprint protocol, specifically in two-week sprints. Furthermore, Agile was chosen due to its flexible structure and its environment allowing a virtual environment. This will help avoid possible conflicting schedules as well as unanticipated black swan events. There will also be meetings twice a week virtually over Discord. In addition, organization is essential when working in sprints. The team will use organizational tools such as JIRA to coordinate task management.

Development Timeline

The timeline will follow over the course of the semester. Each sprint will roughly take two weeks so there will be around seven sprints for this project. There will be two teams: Team A (algorithms focused) and Team B (UI/Unity focused). These teams will mostly work independently so there will be one big sprint where there will be tasks assigned to the specified teams.

Sprints	**Algorithm Team**	**Unity Team**
Sprint 1	Represent the cube programmatically as a 2D array.	Implement cube movement with a keyboard, ie R, L, U, D, F, B can make

	Add functions for the different moves to manipulate the array.	the cube move.
Sprint 2	Develop the layer by layer solving algorithm. Define what a move set may look like and be able to return that.	Allow the user to implement their own cube. Use user input to do this.
Sprint 3	Develop the CFOP method, again make sure the function returns a move set.	Work on the UI for the program, change the movement of the cube to be moved from a moveset, one that may be returned from one of the solving algorithms.
Sprint 4	Develop the Thistlethwaite algorithm, again return a move set	Test/Debug the unity scenes, movement, UI buttons, etc.
Sprint 5	Connect the functions to the Unity scripts, so the Unity scripts can have the move set to move the cube.	Begin to use the movesets that are returned by the solving algorithms to be able to move the cube.
Sprint 6	Testing/Bug fixes	Testing/Bug fixes
Sprint 7	Testing/Bug fixes	Testing/Bug fixes

Work Delegation

We plan to have two teams through the course of development, the Algorithms team and the Unity team. The algorithm team will handle all traditional backend tasks. They will develop the programmatic model of the cube, all functions to interact with the cube including rotations, and ensure that moves are efficient. They will also implement each cube solving algorithm we have developed, Layer-By-Layer, CFOP, and Thistlethwaite's algorithm and ensure that they run accurately and efficiently. The Unity team will develop all traditional frontend tasks. They will work primarily in Unity developing the 3D

model of the cube, and animations of the cube. They will also create and refine the user interface of the program, and ensure that cube rotations are smooth and accurate. These teams are subject to change based on the needs of each task.

Algorithm Team	Unity Team
Noah Backman	Ryan Bell
Tyler Wilkins	Jaylen Thomas
Noah McDonell	
Andrew Li	

Group Member	Assigned Tasks
Noah Backman	Implement Thistlethwaite's Algorithm, connect algorithm to Unity
Tyler Wilkins	Represent movements of the cube programmatically, implement Layer-By-Layer and CFOP algorithms, connect algorithms to the rest of the system
Noah McDonell	Represent the cube programmatically as a 2D array, cube mechanics
Andrew Li	Represent the cube programmatically as a 2D array, cube mechanics, connect the scripts, algorithm implementation
Ryan Bell	Connect the solving algorithms to the C# scripts, Change the movement of the cube to the movesets that are returned from the solving algorithms.
Jaylen Thomas	Implement cube movement with a

	keyboard, ie R, L, U, D, F, B can make the cube move, Connect user input to Unity, Create Unity Scenes and animate movesets

Possible Troubles and Contingency Planning

Task taking longer than expected

If a task for a specific team is taking too long, or the team is falling behind pace, one solution is to pull a member from the other team to help temporarily. This is especially useful if the other team is ahead, but it may not always be an option. If both teams are falling behind, or one can not lose a member of the group at that time, we may need to push some tasks on the timeline back. Hopefully, the team could catch up later on a different sprint. If needed, some goals or tasks of lesser importance could be scrapped in the final weeks if we fall behind schedule. Thus, we want to schedule the most important tasks earlier in the development process.

Needing resources from other team

Over the course of this project it is likely that at some point one of the teams will fall behind the pace of the other. To account for this, the boundary between teams will be flexible. If an issue shows itself to be more complex than expected, that team is welcome to request help or a transfer of a member from the other team. As well, in the case that a team completes their issues prior to the end of the sprint, that Team will

transfer to helping the other team for the remainder of the sprint rather than taking on more issues from the backlog.

Unexpected algorithm result

Testing the algorithm as we develop it will be extremely important to ensuring the correctness of each algorithm. However, the algorithms could develop other issues, such as the possibility of them taking too long (minutes rather than seconds) to deliver a solution. This might require figuring out ways to speed up the algorithm or finding a different one entirely. Other algorithms that could be used to solve the cube include:

- Kociemba's algorithm
- Petrus method
- Roux method

Possibility of Python for Unity no longer being supported

For the scope of this project and what our team is most experienced in, Python would be the best option for developing the algorithms. Luckily, there is a 3rd Party Software called Python For Unity, which has an In-Process API that allows you to invoke Python code from C# and run it in the Unity Process. With this, we can run the algorithms in Python and output whatever data we need to Unity. From there, we will use that data to control the animations, which will be handled in C#. However, since Python For Unity is a 3rd party software we can't be sure that it will have continual support, so we must have a contingency plan for that. Our plan is to use C# instead for

all operations. This wouldn’t be too much of an obstacle, as a couple of our members have expressed that they have experience in it.

Works Cited

[1] "System Requirements for Unity 2020.1." *Unity*, Unity Technologies, 24 Feb. 2021, https://docs.unity3d.com/2020.1/Documentation/Manual/system-requirements.html.

[2] Fourment, Mathieu, and Michael R Gillings. "A Comparison of Common Programming Languages Used in Bioinformatics - BMC Bioinformatics." *BioMed Central*, BioMed Central, 5 Feb. 2008, https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-82.

[3] Sivan, Vishnu. "Running Your Python Code in Unity." *Medium*, Geek Culture, 21 July 2021, https://medium.com/geekculture/running-your-python-code-in-unity-163272682e5

[4] "Introduction to Tensorflow." *TensorFlow*, https://www.tensorflow.org/learn.

[5] Technologies, Unity. "Unity - Manual: System Requirements for Unity 2021 LTS." *Unity*, docs.unity3d.com/Manual/system-requirements.html. Accessed 12 May 2022.

[6] Duffy, Sean. "Using Meshroom to Insert Real Life Objects in Unity." *Raywenderlich.Com*, 5 Aug. 2020, www.raywenderlich.com/9559662-using-meshroom-to-insert-real-life-objects-in-unity.

[7] Hubert, Ian. "About Blender." *Blender*, www.blender.org/about. Accessed 4 Dec.

2022.

[8] N/A. “Comparing Different Methods for Solving the Rubik's Cube.” *GoCube*, 1 Apr. 2021, https://getgocube.com/learn/different-methods-solving-rubiks/.

[9] N/A. “Layer by Layer.” *Layer by Layer - Speedsolving.com Wiki*, 9 Dec. 2020, https://www.speedsolving.com/wiki/index.php/Layer_by_layer.

[10] N/A. “CFOP Method.” *CFOP Method - Speedsolving.com Wiki*, 17 Jan. 2022, https://www.speedsolving.com/wiki/index.php/CFOP_method

[11] N/A. “Roux Method.” *Roux Method - Speedsolving.com Wiki*, 25 Dec. 2021, https://www.speedsolving.com/wiki/index.php/Roux_method.

[12] N/A. “ZZ Method.” *ZZ Method - Speedsolving.com Wiki*, 17 Jan. 2022, https://www.speedsolving.com/wiki/index.php/ZZ_method.

[13] Korf, Richard E. “Finding Optimal Solutions to Rubik's Cube Using Pattern Databases.” *The Association for the Advancement of Artificial Intelligence*, 1997, https://www.aaai.org/Papers/AAAI/1997/AAAI97-109.pdf.

[14] Botto, Ben. “Implementing an Optimal Rubik's Cube Solver Using Korf's Algorithm.” *Medium*, Medium, 10 Jan. 2022, https://medium.com/@benjamin.botto/implementing-an-optimal-rubiks-cube-solver-using-korf-s-algorithm-bf750b332cf9.

[15] Scherphuis, Jaap. “Thistlethwaite's 52-Move Algorithm.” *Jaap's Puzzle Page*, https://www.jaapsch.net/puzzles/thistle.htm#p26. .

[16] “The Devils Number - History of the Cube Mystery.” *The Devils Number - History of the Cube Mystery*, 29 May 2020, https://getgocube.com/play/devils-number/#:~:text=The%20%E2%80%9CDevil's%20Algorithm%E2%80%9D%20is%20an,along%20the%20sequence%20of%20moves.

[17] Lippman, David. “Mathematics for the Liberal Arts Corequisite.” *Lumen*, https://courses.lumenlearning.com/mathforliberalartscorequisite/chapter/hamiltonian-circuits/.

[18] N/A. “How to Solve the Rubik's Cube.” *How to Solve the Rubik's Cube - Beginners Method*, https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/.